Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 25

Space and Time Scalability of Duplicate Detection in Graph Data

Melanie Herschel und Felix Naumann

Potsdam 2008

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de/ abrufbar.

Universitätsverlag Potsdam 2008
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: -4625
E-Mail: ubpub@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik and der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Sie erscheint aperiodisch.

Autoren: Melanie Herschel, Felix Naumann

Das Manuskript ist urheberrechtlich geschützt.
Druck: allprintmedia GmbH, Berlin

ISSN 1613-5652
ISBN 978-3-940793-46-1

Space and Time Scalability of Duplicate Detection in Graph Data

Melanie Herschel and Felix Naumann
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
(fistname.lastname)@hpi.uni-potsdam.de

Abstract

Duplicate detection consists in determining *different* representations of real-world objects in a database. Recent research has considered the use of relationships among object representations to improve duplicate detection. In the general case where relationships form a graph, research has mainly focused on duplicate detection quality/effectiveness. Scalability has been neglected so far, even though it is crucial for large real-world duplicate detection tasks.

In this paper we scale up duplicate detection in graph data (DDG) to large amounts of data and pairwise comparisons, using the support of a relational database system. To this end, we first generalize the process of DDG. We then present how to scale algorithms for DDG in space (amount of data processed with limited main memory) and in time. Finally, we explore how complex similarity computation can be performed efficiently. Experiments on data an order of magnitude larger than data considered so far in DDG clearly show that our methods scale to large amounts of data not residing in main memory.

Chapter 1

Preliminaries

1.1 Introduction

Duplicate detection is the problem of determining that different entries in a database actually represent the same real-world object, and this for all objects represented in a database. Duplicate detection is a crucial task in data cleaning [15, 25], data integration [11], and many other areas. The problem has been studied extensively for data stored in a single relational table. However, much data comes in more complex structures. For instance, a relational schema is not limited to a single table, it also includes foreign keys that define relationships between tables. Such relationships provide additional information for comparisons.

Recently, a new class of algorithms for duplicate detection has emerged, to which we refer to as *duplicate detection in graphs (DDG)* algorithms. These algorithms detect duplicates between object representations in a data source, called *candidates* by using relationships between candidates to improve effectiveness. Within this class, we focus on those algorithms that iteratively detect duplicates when relationships form a graph (see Sec. 1.2). In the remainder of this paper, reference to DDG algorithms imply the restriction to that particular class of algorithms. To the best of our knowledge, none of the proposed algorithms within this class has explicitly considered scalability yet.

Before delving into the details of scaling up DDG, we provide readers with an intuition of how DDG algorithms behave based on the following example.

Example 1 *Figure 1.1 represents a movie database consisting of three tables, namely MOVIE, ACTOR, and STARS_IN. STARS_IN translates the m:n relationship between movies and actors. Key and foreign key definitions are straightforward and are not represented for conciseness. It is obvious to humans that all three movies represent a single movie and there are only three distinct actors, despite slightly different representations.*

Assume that we consider movies, titles, and actors as candidates, which we identify by an ID for ease of presentation and future reference. Title candidates illustrate that candidates are not necessarily represented by a single relation. We use ' to mark duplicates, e.g., `m1` *and* `m1'` *signifies that the movies are duplicates.*

Initialization. *DDG algorithms usually represent the data they consider as a graph. Figure 1.2 depicts a possible graph for our movie scenario. In this graph, there is one candidate node*

MID	title
m1	Troy
m1’	Troja
m1”	Illiad Project

(a) MOVIE

AID	Name
a1	Brad Pitt
a2	Eric Bana
a1’	Brad Pit
a2’	Erik Bana
a3	Brian Cox
a1”	Prad Pitt
a3’	Brian Cox

(b) ACTOR

AID	MID
a1	m1
a2	m1
a1’	m1’
a2’	m1’
a3	m1’
a1”	m1”
a2’	m1”

(c) STARS_IN

Figure 1.1: Sample movie database

for each candidate, whereas its descriptive information, given by the attributes of the relational table are represented as attribute nodes associated with the corresponding candidate note. Candidate nodes are connected to each other by directed edges, called dependency edges. Intuitively, these edges represent the fact that finding duplicates of the source candidate depends on finding duplicates of the target candidate. The latter are called influencing candidates.

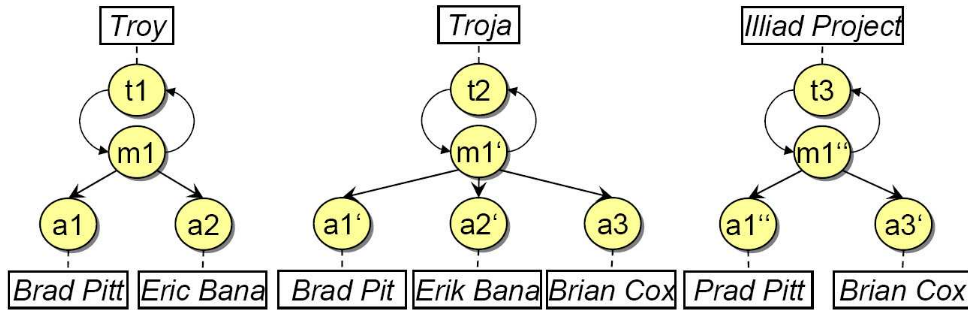


Figure 1.2: Sample reference graph. ◯ candidate node, □ attribute node, ↓ dependency edge.

Next, DDG algorithms initialize a priority queue of candidate pairs. Each pair in the queue is compared based on some similarity measure that considers both neighboring attribute nodes and influencing candidate nodes. If the similarity is above a given threshold, the candidate pair is considered as a duplicate. Assume that the initial priority queue is PQ = { {(m1, m1’), (m1, m1”), (m1’,m1”), (t1,t2), (t1,t3), (t2,t3), (a1,a2),...} }.

Iterative phase. *We now start comparing pairs in the priority queue. The first pair retrieved from PQ is (m1, m1’). The pair is classified as non-duplicate, because the movies’ sets of influencing neighbors, i.e., the respective actor neighbors are apparently all non-duplicates. The same occurs when subsequently iterating through movie candidate pairs and title candidate pairs. When comparing (a1,a1’), we finally find a duplicate because the actor names are similar, and there are no influencing neighbors. Having found this duplicate potentially increases the similarity of (m1, m1’) because the movies now share actors. Hence, (m1, m1’) is added back to PQ. The same occurs in the subsequent iterations where duplicate actors are detected. Now, when we compare movies again they can be classified as duplicates, because they share a significant amount of influencing actor neighbors. In the end, we identify all duplicates*

correctly.

We make the following observations: First, re-comparisons, e.g., of the movie candidates, increase effectiveness and should therefore be performed. However, if we had started by comparing actors, we would have avoided classifying movies a second time and would still have obtained the same result. This shows that the comparison order is important for efficiency. Hence, the order of the priority queue should be chosen carefully when initializing the priority queue and it should adapt to updates of the priority queue, e.g., when a movie pair is added back to the priority queue. Finally, this process should scale to large amounts of data beyond main memory. Whereas effectiveness and priority queue order maintenance for efficiency have been considered in the past, scalability of DDG has not been addressed so far. This paper aims at filling this gap.

More specifically, we present how *iterative algorithms for DDG* can scale up to large amounts of data. We use a relational DBMS as storage manager and overcome its limitations in supporting DDG. Our contributions are: (i) A generalization of iterative DDG, (ii) an algorithm for scaling DDG data retrieval and update in space and in time, and (iii) methods to scale similarity computation in space and in time. All methods proposed for scalability are based on our generalization of iterative DDG, thus making them applicable to existing iterative DDG algorithms, which are so far limited to main-memory processing. Our most significant results include successful DDG over 1,000,000 candidates, a data set an order of magnitude larger than any data set considered so far, and a close to linear behavior of DDG runtime with respect to several parameters.

We discuss related work in Sec. 1.2. We then generalize iterative DDG and provide definitions in Sec. 1.3. Section 2.1 describes how initialization of the algorithm is performed when main-memory does not suffice to complete DDG, and Sec. 2.2 presents how we scale up the iterative phase of DDG. Similarity computation is discussed in Sec 2.3. Evaluation is provided in Sec. 3 before we conclude in Sec. 4.

1.2 Related Work

The problem of identifying multiple representations of a same real-world object, originally defined in [22], and formalized in [14] has been addressed in a large body of work. Ironically, it appears under numerous names, e.g., record linkage [18], merge/purge [17], object matching [15], object consolidation [10], and reference reconciliation [12]. A duplicate detection survey is provided in [13].

Broadly speaking, research in duplicate detection falls into three categories: techniques to improve *effectiveness*, techniques to improve *efficiency*, and techniques to enable *scalability*. Research on the first problem is concerned with improving precision and recall, for instance by developing sophisticated similarity measures [7, 9] or by considering relationships [28]. Research on efficiency assumes a given similarity measure and develops algorithms that avoid applying the measure to *all pairs* of objects [1, 21]. To apply methods to very large data sets, it is essential to scale not only in time by increasing efficiency, but also to scale in space, for which relational databases are commonly used [17, 24].

	Table	**Tree**	**Graph**
Learning	Bil.03[7](Q) Sar.02[27](Q)		Sin.05[28](Q) Bhat.06 [5](Q)
Clustering	Chau.05[9](Q,T,S)		Kala.06[19, 10](Q, T) Yin06[33](Q,T)
Iterative	Her.95[17](T,S) Mon.97[21](T) Jin03[18](Q,T)	Ana.02[1](Q,T,S) Puh.06[24](T,S) Mil.06 [20](Q)	Dong05[12](Q) Weis06[30, 31](Q,T) Bhat.04[2, 3, 4](Q)

Table 1.1: Summary of duplicate detection approaches, their focus lying on efficiency(T), effectiveness(Q), or scalability(S).

In Tab. 1.1, we summarize some duplicate detection methods, classifying them along two dimensions, namely *data* and *approach*. For data, we distinguish between (i) data in a single *table*, without multi-valued attributes, (ii) *tree* data, such as data warehouse hierarchies or XML data, and (iii) data represented as a *graph*, e.g., XML with keyrefs or data for personal information management (PIM). The second dimension discerns between three approaches used for duplicate detection: (i) machine *learning*, where models and similarity measures are learned, (ii) the use of *clustering* techniques, and (iii) *iterative* algorithms that iteratively detect pairs of duplicates, which are aggregated to clusters. In Tab. 1.1, we also show whether an article mainly focuses on efficiency (T), effectiveness (Q), or scalability(S). Due to space constraints, we limit the discussion to approaches falling in the class of algorithms we consider, i.e., iterative duplicate detection algorithms in graph data.

Research presented in [12] performs duplicate detection in a PIM scenario, where relationships between persons, publications, etc. form a graph. Pairs of candidates to be compared are maintained in an active priority queue where the first pair is retrieved at every iteration. The pair is compared and classified as non-duplicate or duplicate. In the latter case, relationships are used to propagate the decision to neighbors, which potentially become more similar due to the found duplicate, and the priority queue is reordered, neighbors being sent to the front or the back of the queue. The largest data set consists of 40,516 candidates, but no runtime are reported.

In [30, 31], we presented similar algorithms to [12], and focused on the impact of comparison order and recomparisons on the efficiency and effectiveness of DDG. We showed that the comparison order can significantly compromise efficiency and effectiveness in data graphs with high connectivity. For evaluation, we used small data sets ($\approx$ 2,500 candidates) which easily fit in main-memory.

The RC-ER algorithm first introduced in [3] and further described and evaluated in [2, 4] represents candidates and dependencies in a reference graph. The algorithm re-evaluates distances of candidate pairs at each iterative step, and selects the closest pair according to the distance measure. Duplicates are merged together before the next iteration, so effectively clusters of candidates are compared. The largest data set [3] contains 83,000 candidates and duplicate detection takes between 310 and 510 seconds, depending on which variant of the algorithm is used. [4] describes the use of an indexed max-heap data structure to maintain the order of the

priority queue, but this structure again fits in main memory. Experiments in [2] show that the higher connectivity of the reference graph, the slower RC-ER.

The focus of DDG has so far been set on effectiveness. Scaling DDG to large amounts of data not fitting in main memory has not been considered at all, and efficiency has been only a minor concern. This gap is filled by the research presented in this paper.

1.3 DDG in a Nutshell

The methods we propose to scale up DDG are intended to be as general as possible to fit many algorithms performing iterative DDG. Therefore, the first contribution of this paper is a generalization of existing methods into an unified model.

1.3.1 Definitions for unified DDG

Before defining our unified model, we make the following definitions that are used both to define the model and to describe our methods for scaling up DDG.

Candidates. The representations of real-world objects subject to duplicate detection are called *candidates*. A database represents several object types $T = \{t_1, t_2, ..., t_n\}$. Let $C_t = \{c_1, c_2, ..., c_k\}$ be the set of candidates of type $t \in T$. To denote the type of a particular candidate c, we use t_c. In relational tables, candidates correspond to (projections of) tuples.

Thresholded Similarity Measure. During classification, DDG algorithms commonly use a similarity measure. A more detailed description of a similarity measure template is given in Sec. 2.3.1 when discussing methods to scale up classification. At this point, we only want to highlight that, given a similarity measure $sim(c, c')$ and a threshold θ, we conclude that c and c' are duplicates if $sim(c, c') > \theta$, and non-duplicates otherwise. Intuitively, the similarity measure both considers object descriptions and influencing candidate pairs in its computation, which we define next.

Object Descriptions. The *object description (OD)* of a candidate that corresponds to a tuple consists of attributes and attribute values of this tuple. Let $A_t = \{a_1, ..., a_l\}$ be a set of attribute names for a candidate of type t. Then, $OD_a(c) = \{v_1, ..., v_m\}$ defines the attribute values of the attribute named a that are part of the c's OD. The complete OD of a candidate c is defined as $OD(c) = \bigcup_{a_i \in A_t} OD_{a_i}(c)$, and is usually obtained using a query parameterized by the candidate.

Example 2 *In our example, consider candidate types $T = \{movie, actor, title\}$. A candidate of type actor is described by it's name, so the OD of actor candidate* a1 *is $OD(\textsf{a1}) = OD_{Name}(\textsf{a1}) = \{(\textsf{Brad Pitt})\}$.*

Influencing & Dependent Candidates. The set of *influencing candidates* $I(c)$ of a candidate c is a set of candidates that has been designated to affect the similarity of c as follows: if the similarity (or duplicate status) of a candidate $i \in I(c)$ to another candidate changes, then the similarity (or duplicate status) of c to another candidate changes as well. Which other candidates are involved in this dependency is defined next. The selection of candidates in $I(c)$ is for instance based on foreign key constraints, or implicit domain constraints provided by

a domain expert. Analogously, *dependent candidates* of c are candidates whose similarity is influenced by c, i.e., $D(c) = \{c' | c' \neq c \wedge c \in I(c')\}$.

Example 3 *We assume* $I($m1$) = \{$a1, a2, t1$\}$ *and* $D($m1$) = \{$t1$\}$. *Similarly,* $I($m1"$) = \{$a1", a3', t3$\}$ *and* $D($m1"$) = \{$t3$\}$.

Influencing & Dependent Candidate Pairs. *Influencing candidate pairs* of candidate pair (c, c') are defined as $I(c, c') = \{(n_i, n_j) | n_i \in I(c), n_j \in I(c'), t_{n_i} = t_{n_j}\}$. Analogously, its *dependent candidate pairs* $D(c, c') = \{(n_i, n_j) | n_i \in D(c), n_j \in D(c'), t_{n_i} = t_{n_j}\}$. Intuitively, influencing and candidate pairs of two candidates are obtained by forming the cross product between their respective influencing and dependent candidates.

Example 4 *We assume* $I($m1, m1"$) = \{$(a1, a1"), (a1, a3'), (a2, a1"), (a2, a3'), (t1, t3)$\}$, *and* $D($m1, m1"$) = \{$(t1, t3)$\}$.

Based on these definitions, we now define a unified view of iterative DDG algorithms.

1.3.2 General DDG Approach

In general, we observe that algorithms for iterative DDG have in common that (i) they consider data as a graph, (ii) they perform some preprocessing before candidates are compared to obtain initial similarities and avoid recurrent computations, and (iii) they compare pairs of candidates iteratively in an order that potentially changes at every iteration where a duplicate is found, requiring the maintenance of an ordered priority queue. Merging [3] or enriching [12] detected duplicates is also a common technique to increase effectiveness and requires updating the graph. Based on these observations, we devise the following general approach for DDG.

Unified Graph Model. Existing DDG approaches all have in common that they use a graph to represent the data. [12] uses a *dependency graph* whose nodes are pairs of candidates $m = (r_1, r_2)$ and pairs of attribute values $n = (a_1, a_2)$, and an edge between m and n exists if a_1 is an attribute value of r_1 and a_2 is an attribute value of r_2. Further edges between candidate pair nodes m_1 and m_2 exist if m_1 is an influencing or dependent pair of m_2. The method used in [3] defines a *reference graph*, where nodes do not represent pairs, but candidates and edges relate influencing or dependent candidates. Other DDG algorithms use graphs that basically fall in one of the two categories described: nodes either represent single candidates [19, 30], or pairs [28]. Attribute values are usually treated separately. In the context of duplicate detection, the dependency graph can be derived from the reference graph[1]. Hence, our unified graph model for DDG is a reference graph that consists of three components, namely (i) candidate nodes, (ii) attribute nodes, and (iii) dependency edges. More specifically, a candidate node v_c is created for every candidate. Every attribute value associated with a candidate is translated by an attribute node v_a. We represent relationships between candidates as directed edges in the graph. More specifically, let v_c and v_c' be two candidate nodes. We add an edge $e_d = (v_c, v_c')$ directed from v_c to v' if $v_c' \in I(c)$. Figure 1.2 is a possible reference graph for our movie example.

[1] In general, a dependency graph is more expressive, but its full expressiveness is not used in DDG.

Unified DDG Initialization. To detect duplicates, DDG algorithms set up a *priority queue* PQ where candidate pairs are sorted according to a specific ordering scheme. Note that blocking [21] or filtering [1] methods can be used to restrict the candidate pairs entering PQ. Formally, the set of pairs in PQ is defined by

$$PQS \subseteq \bigcup_{1 \le i \le n} \{(c, c') | v_c \in C_{t_i} \wedge v_{c'} \in C_{t_i} \wedge c \neq c' \wedge t_i \in T\}$$

Furthermore, iterative DDG algorithms have a jump-start phase, where attribute similarities are computed or obvious duplicates are detected. Also, priority queue initialization and precomputations are performed before candidate pairs are compared and classified and thus compose the *initialization phase* of a DDG algorithm.

Iterative Phase. After initialization, candidate pairs are compared and classified as duplicates or non-duplicates in the *iterative phase* (Fig. 1.3). Existing algorithms maintain PQ in main-memory. At every iteration step, the first pair in PQ is *retrieved*, then *classified* using a similarity (distance) metric, and finally the classification causes some *update* in PQ or the graph (adding pairs to PQ, enrichment in [12], duplicate merging and similarity recomputation in [3]) before the next iteration starts.

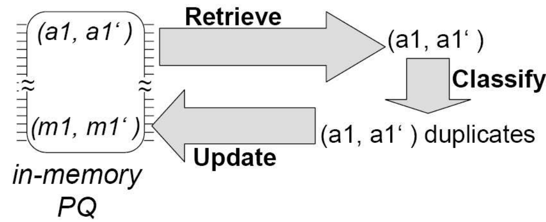


Figure 1.3: Sample iteration for DDG

In general, the priority queue has to be reordered whenever a duplicate is detected to reduce the number of recomparisons, for which [3, 12, 30] devise different strategies. Reordering is an expensive task for large PQ. However, by maintaining the order in PQ, complex similarity computations are potentially saved, as we illustrated in Example 1.

Chapter 2

Scaling up DDG

2.1 Scaling Up Initialization

Having discussed the general DDG process, we now show how initialization is performed when processing large amounts of data. To scale up the initialization phase, we use a relational DBMS to store the data structures described in the next subsections. This way, DDG can process an arbitrarily large amount of data and can benefit from several DBMS features, such as access to the data, indices, or query optimization. Essentially, the DBMS acts as a data store.

2.1.1 Graph Model in Database

The reference graph G_{ref} does not fit in main-memory when considering large amounts of data, so we decide to store G_{ref} in a relational database. For every candidate type $t \in T$ a relation *C_t(CID,* a_1*,* a_2*, ...,* a_n*)* is created. *CID* represents a unique identifier for a candidate c, and $a_i \in A_t$ represents the candidate's OD attributes. Hence table *C_t* stores both the information contained in candidate nodes and the information contained in their related attribute nodes. Using standard SQL statements, the tables are created in the database and tuples are inserted as follows. For every candidate c of type t, e.g., returned by a SQL query, determine $OD(c)$, given the set of OD labels $A_t = \{a_1, ..., a_l\}$, again using an extraction method that can be a query language or a parsing program. We then create a tuple <$id(c)$, $v_1 \in OD_{a_1}(c)$, ..., $v_l \in OD_{a_l}(c)$> for every candidate, $id(c)$ being a function that creates a unique ID for a candidate. If OD_{a_i} contains more than one value, values are concatenated using a special character.

To store edges in the database, we create a single table *EDGES(S,T,REL)*: the first attribute *S* is the source candidate, *T* is the target candidate, and *REL* is the relevance (weight) of that edge. Both the source and the target attribute are references to a *CID* in a candidate table.

Example 5 *Fig. 2.1 shows excerpts of data graph tables for our movie example. There is one table for every candidate type, i.e., one for movie candidates (a), one for title candidates (b) and one for actor candidates (c). The table representing RDs is shown in Fig. 2.1(d). In our example, all edges have equal relevance, set to 1.*

CID
m1
m1'
m1"

(a) C_MOVIE

CID	OD1
t1	Troy
t2	Troja
t3	Illiad...

(b) C_TITLE

CID	OD1
a1	Brad Pitt
a2	Eric Bana
...	...

(c) C_ACTOR

S	T	REL
m1	a1	1
m1	a2	1
...	...	...

(d) EDGES

Figure 2.1: Graph representation in database

2.1.2 Initializing the Priority Queue

The priority queue used in DDG is a sequence of candidate pairs, where two candidates of the same pair have same candidate type and the priority queue contains all candidate pairs subject to comparison. The order of pairs is determined by an algorithm-dependent heuristic that satisfies the following property: let $(c, c') \in PQ$, and let the position of $(c, c') = rank_i(c, c')$ at iteration i, then the heuristic guarantees that when a pair $(i, i') \in I(c, c')$ is classified as duplicate, $rank_{i+1}(c, c') \leq rank_i(c, c')$ at the next iteration. PQ is ordered in ascending order of $rank$. Using such a rank function can implement the ordering in [3, 12, 30], and is natural, because the similarity of pairs increases with increasing number of similar influencing neighbors, and more similar pairs should be compared first, because they are more likely to be classified as duplicate.

In the database, we store the priority queue PQ in a relation *PQT(C1, C2, STATUS, T, RANK)*, where *C1* and *C2* are references to the *CID* of two candidates of a pair, *T* is both candidate's type, *RANK* is the current rank of a pair, which determines its position in the priority queue order, and *STATUS* is the processing status of the pair in the specific DDG algorithm. For instance, *STATUS* = 0 if a pair is in the priority queue and has not been classified, *STATUS* = 1 if the pair is a duplicate, and *STATUS* = -1 if the pair has been classified as a non-duplicate but may re-enter the priority queue if an influencing duplicate were found.

Example 6 *Examples of tuples in table PQT are shown in Fig. 2.2. As sample ranking heuristic, we use the number of unshared influencing neighbors between the two candidates.*

C1	C2	STATUS	T	RANK
m1	m1'	0	M	7
a1	a2	0	A	0
t1	t2	0	T	2
...	...	...	...	...

Figure 2.2: Initial priority queue sample

2.1.3 Precomputations

As we will see in Sec. 2.3, the similarity measure we consider requires the computation of both OD similarity and OD difference. Furthermore, description values can in general have a

A	W
Brad Pitt	1
Eric Bana	1
...	...

(a) *ODW_actor_OD1*

A1	A2	W
Brad Pitt	Brad Pit	1
Eric Bana	Erik Bana	1
...	...	...

(b) *ODSIMW_actor_OD1*

S1	S2	T1	T2	STATUS_S	STATUS_T	W_S	W_T
m1	m1'	a1	a1'	0	0	1	1
m1	m1'	a1	a2'	0	0	1	1
...	...	...	...	...	...	...	...

(c) *DEP*

Figure 2.3: Sample precomputation tables

weight that is considered in similarity measurement. These values do not vary during the iterative phase, so we decide to compute them prior to the iterative phase. This way, we potentially save expensive and recurrent similarity computations. Indeed, the same computations may be performed several times due to recomparisons. Moreover, when using functions such as the inverse document frequency of a value for weight determination or the edit distance between two values for OD similarity, the function depends on the value of a particular description, not on the candidate to which the OD belongs to. Hence, if the same attribute value occurs several times in different pairs, the computation would be performed more than once. Following these observations, we precompute (i) weights of attribute values, which we store in *ODW_a_t(A,W)* relations, i.e., one relation for each OD attribute and (ii) weights of similar OD value pairs, which are stored in *ODSIMW_a_t(A1,A2,W)* relations. Note that *A*, *A1*, *A2* designate OD attribute values, and *W* stores the weight.

Another precomputation is the creation of a table that associates candidate pairs with their influencing pairs and that stores the duplicate status of both pairs, as well as the weight of their relationships. This way, the comparison of influencing neighbor sets, which is also part of the similarity measures can be computed incrementally and without an expensive join over *PQT* and *EDGES* that would otherwise be necessary. The schema of the table is *DEP(S1,S2,T1,T2,STATUS_S,STATUS_T,W_S,W_T)*, where S1 and S2 describe a candidate pair, and *T1* and *T2* its influencing pair. The respective duplicate status of each candidate pair is stored in the *STATUS* attributes, whereas weights are stored in *W* attributes.

Example 7 *Fig. 2.3 shows a table excerpt of the three types of precomputations we perform. Fig. 2.3(a) shows precomputed weights for actor values, Fig. 2.3(b) shows weights of similar attribute values, and Fig. 2.3(c) shows the table storing candidate pair relationships.*

How to perform precomputation efficiently is not discussed in this paper, as the approach is specific to the particular similarity measure or DBMS used. As example, [16] discusses how to compute string similarity joins efficiently in a database. In [8], an operator for similarity joins for data cleaning is presented. We focus on scaling up the unified iterative phase both in space and in time.

2.2 Scaling Up Retrieval & Update

As described in Sec. 1.3.2, the iterative phase in DDG consists of a retrieval step, a classification step, and an update step. In this section, we focus on scaling up retrieval and update, and postpone the discussion of classification to Sec. 2.3 for which we present solutions orthogonal to the choice of the retrieval and update algorithm.

A straightforward approach to scale up DDG is mapping the DDG process from main-memory (Fig. 1.3) to a database. We refer to this baseline algorithm as RECUS/DUP, as the **Re**trieval-**C**lassify-**U**pdate-**S**ort process is guided by duplicate classifications. Although being straightforward, we discuss RECUS/DUP, because it is the basis for our algorithm that scales DDG both in space and in time. Algorithm 1 provides pseudocode for RECUS/DUP.

RECUS/DUP Retrieval Phase. Candidate pairs are retrieved from the database as long as the priority queue table *PQT* contains non-duplicate candidate pairs, i.e., pairs having *STATUS* = 0. To retrieve pairs, we send a query to the database, which returns unclassified pairs in ascending order of their *RANK*. As long as no duplicate is found according to the *classify*() function (discussion postponed to Sec. 2.3), the order of candidate pairs in the priority queue remains the same, so we simply iterate over these pairs. When a duplicate is detected, the retrieval query is issued again at the next iteration and we classify pairs returned by the new result. Thus, we sort the data stored in table *PQT* during the retrieval phase only when it is necessary, i.e., when a duplicate is found and the rank of dependent candidate pairs potentially changes.

RECUS/DUP Update Phase. After a pair has been classified, the status of the retrieved pair is set to 1 if it has been classified as a duplicate, and to -1, if it has been classified as non-duplicate. Note that the function *updateValue*() updates both the value in the ResultSet as well as the value in the database, i.e., in *PQT*.

In case of a duplicate classification, duplicates can be merged, enriched, or clustered in any other way using the *cluster*() function, which updates graph tables and precomputed tables according to the algorithm-specific method. The next step in any algorithm is to update the status of the duplicate pair in table *DEP*. Finally, the status of every non-duplicate dependent pair has to be reset to 0, because of new evidence that they may be duplicates.

RECUS/DUP makes the least possible use of main-memory by keeping only a single pair and possibly a dependent pair in main memory at every iteration, plus some information to compute similarity. Thereby, RECUS/DUP can be applied to arbitrary large amounts of data. However, as experiments show (see Sec. 3), sorting *PQT* is a very time consuming task. The algorithm described next, namely RECUS/BUFF, reduces the sorting effort of RECUS/DUP and makes DDG more efficient by using main memory more extensively, but with an upper bound.

RECUS/BUFF uses an in-memory buffer B_s of fixed size s to avoid sorting *PQT* each time a duplicate is found. The intuition behind RECUS/BUFF is that although ranks of several pairs may change after a duplicate has been found, sorting *PQT* immediately after finding the duplicate is not always necessary and may actually occur several times before an affected pair is retrieved. For instance, consider the initial priority queue order of Fig. 2.1(e). All candidate actor pairs have rank 0. When a1 and a1' are detected to be duplicates, the rank of their dependent neighbor pair (m1,m1') computed as the number of non-duplicate influencing candidates

changes from 7 to 5. Although the rank has decreased, all actor and title pairs still have a lower rank (0 and 2, respectively), and will be compared first. Hence, sorting the priority queue does not immediately affect the comparison order and should therefore be avoided. To this end, we use B_s to temporarily store dependent neighbors of pairs classified as duplicates, whose rank potentially decreases, and maintain these pairs in the order of ascending rank (the same order as the *PQT*), which avoids sorting the much larger *PQT*.

Using an in-memory buffer requires modifications in the retrieval and update phase of RECUS/DUP, because a pair is either retrieved from *PQT* on disk or the buffer B_s in main-memory, and is either updated in *PQT* or in B_s, as depicted in Fig. 2.4.

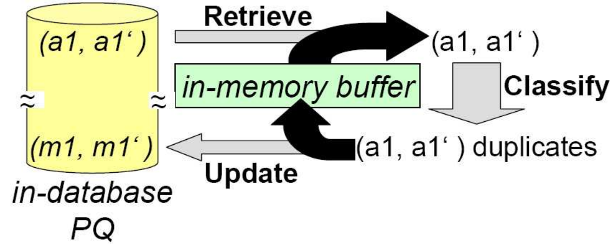


Figure 2.4: Sample RECUS/BUFF iteration

RECUS/BUFF Retrieval Phase. In Alg. 2, we describe the retrieval phase of RECUS/BUFF. As long as the buffer B_s does not overflow, we distinguish the following two cases, using the $lastFromPQ$ flag: In the first case, the pair to be classified potentially is the next pair from *PQT*. That is, the pair r is pointing to has been classified previously, and we use the *getNextTupel*() function to get the next candidate pair from *PQT*. The remaining steps performed in this case are related to the update phase, discussed further below. In the second case, the pair at the current position of cursor r has not been classified yet, because the first pair in B_s had a lower rank. In this case, we do not move forward the cursor.

RECUS/BUFF Update Phase. Having the correct pair in hand after the retrieval phase shown in Alg. 2, the pair is classified as duplicate or non-duplicate as in RECUS/DUP, before *PQT* and B_s are updated accordingly. The pseudo-code for the RECUS/BUFF update phase is similar to the update phase of RECUS/DUP in Alg. 1, and we highlight only differences. When updating the status of the classified pair, two situations are possible: either the pair has been retrieved from *PQT*, in which case the cursor still points to that pair, or the pair has been retrieved from B_s. In the first case, *STATUS* can directly be updated at the current cursor position of r using *updateValue*(). In the latter case, the current pair, which has been retrieved from B_s, is at a position with higher rank in *PQT*. Hence, the cursor would have to jump forward in r to update the *STATUS* value, and jump back. We avoid this expensive operation by adding or updating the classified pair to B_s, together with its status. The pair stays in B_s until the iterator on r has moved to that pair in *PQT*, in which case it is updated in *PQT* and removed from B_s. We verify whether a pair has been classified and stored in B_s during retrieval and avoid classifying it again by setting the $updateOnly$ flag to true. Instead, we update the status in *PQT* to the status stored in B_s, which contains the most recent status.

Essentially, RECUS/BUFF adds dependent neighbor pairs to B_s instead of updating *PQT*

and the next retrieval phase starts, without sorting the complete data in *PQT*. However, it can be shown that RECUS/BUFF obtains the same final result as RECUS/DUP.

Example 8 *Assume we classify pair (a1,a1') as duplicate before its dependent neighbor pair (m1, m1'). At this point, we add the dependent pair to the buffer and* $B_s = \{$*(m1,m1',5,0)*$\}$*, where 5 is the new rank of the pair and 0 the non-duplicate status. Unlike for* RECUS/DUP*, we do not sort PQT again to get a new ResultSet* r*. Instead, we continue iterating through the same* r*, with outdated rank and order. Now, consider we reach pair (m1, m1") with rank 6 in* r*. When checking for lower rank in* B_s*, we see that it contains a pair with lower rank 5. Hence, the next pair to be classified is the first pair in the buffer, e.g., (m1,m1'). We classify the pair as duplicate. Now, because the cursor on* r *does not point to pair (m1,m1'), which is still sorted according to its old rank (7), we put the pair back to* B_s *with new status 1 to indicate it is a duplicate. It will be updated later in PQT when the cursor on* r *reaches it. At the next iteration, the pair the cursor of* r *points to, e.g., (m1, m1") has not been classified yet, so we do not move the cursor forward.*

In case of a buffer overflow, a strategy that frees buffer space has to be devised. To minimize the occurrences of a buffer overflow, and hence of sorting *PQT*, we clear the entire buffer only when it is full. That is, we update all pairs in *PQT* that were buffered, remove them from B_s, and sort *PQT* in the next retrieval phase.

As for RECUS/DUP, the theoretical worst case requires sorting *PQT* after every iteration. However, whereas RECUS/DUP reaches the worst case when every pair is classified as a duplicate, RECUS/BUFF requires the buffer to overflow at every iteration, an unlikely event especially when wisely choosing the size of the buffer. In setting the buffer size one should keep in mind that (i) it must fit in main-memory, (ii) it should be significantly smaller than table *PQT* to make sorting it more efficient than sorting *PQT*, and (iii) it should be large enough to store a large number of dependent pairs to avoid sorting *PQT*.

In this section, we have discussed the retrieval phase and the update phase of two algorithms that enable scaling DDG both in space and in time. Next, we discuss how to scale up classification.

2.3 Scaling Up Classification

Classifying a candidate pair requires the computation of a similarity that considers both object descriptions (ODs) and influencing candidates (see Sec. 1.3.1). We present a similarity measure template in Sec. 2.3.1. When the reference graph cannot be held in main-memory, similarity computation requires communicating with the database. A straightforward solution, described in Sec. 2.3.2 is to use a SQL query to compute the similarity. However, such a solution is limited by the expressive capabilities of SQL or its implementation. For instance, complex weight aggregate functions are rarely supported by commercial RDBMSs, and it is not possible to interfere with the query processing to improve query execution time. Therefore, we propose hybrid similarity computation (Sec. 2.3.3) and the early classification technique (Sec. 2.3.4).

2.3.1 Similarity Measure Template

In this section, we provide a template for a base similarity measure. Different implementations of this template [1, 3, 12, 29] use one or more similarity measures conforming to the template. In the latter case, base similarity measure are for instance combined with sum or multiplication.

First, we define the set $N^{\approx}_{rd}$ of duplicate influencing candidate pairs of a pair of candidates (c, c') as

$$N^{\approx}_{rd}(c,c') \coloneqq \{(n_1,n_2)|(n_1,n_2) \in I(c,c') \wedge (n_1,n_2) \text{ duplicates}\}$$

The set $N^{\neq}_{rd}$ of non-duplicate influencing candidates is

$$\begin{aligned} N^{\neq}_{rd}(c,c') \coloneqq & \ \{(n_1,\perp)|n_1 \in I(c) \wedge n_1 \text{ has no duplicates in } I(c')\} \\ \cup & \ \{(\perp,n_2)|n_2 \in I(c') \wedge n_2 \text{ has no duplicates in } I(c)\} \end{aligned}$$

where $\perp$ denotes an empty entry. These definitions assume that we know that n_1 and n_2 are duplicates or non-duplicates. This knowledge is acquired during the iterative phase, and the similarity increases as this knowledge is gained.

Example 9 *Assuming that duplicate actor and title candidates have already been detected, we obtain $N^{\approx}_{rd}$(m1, m1”) = $\{$(a1, a1”)$\}$, and $N^{\neq}_{rd}$(m1, m1”) = $\{$(a2, $\perp$), (t1, $\perp$), ($\perp$, a3’), ($\perp$, t3)$\}$.*

We further introduce a weight function $w_{rd}(S)$ that captures the relevance of a set of candidate pairs $S = \{(n_i,n'_i)|n_i \in C_t \wedge n'_i \in C_t, t \in T\}$. This weight function has the following properties allowing its incremental computation and guaranteeing that the complete similarity of two candidates monotonously increases.

$$w_{rd}(S) = WAgg(\bigcup_{(n_i,n'_i)\in S} w_{rd}(\{(n_i,n'_i)\})) \tag{2.1}$$

$$WAgg(\{w_{rd}(\{(n,\perp)\}), w_{rd}(\{(\perp,n')\})\}) \geq w_{rd}(\{(n,n')\}) \tag{2.2}$$

where $WAgg$ is an aggregate function, e.g., *sum* or *count* that combines multiple weights such that $WAgg(S) \geq WAgg(S'), S' \subset S$. The first property guarantees that the weight of a set of pairs can be computed based on weights of individual pairs, allowing an incremental computation of the weight. The second property implies that the aggregated weight of two non-duplicates is larger or equal to their weight if they were duplicates. The second property is reasonable because the relevance of an object represented by two candidates should not be larger than the combined relevance of two individual representations of that object. On the other hand, when we do not know that p and p' are duplicates, we have to assume that they are different and each contribute an individual relevance that is likely to be larger or equal to their combined relevance. More interestingly, this property guarantees that the similarity of two candidates monotonously increases as duplicate descriptions among these candidates are detected. Furthermore, it allows us to estimate the total similarity (with an upper and lower bound) while duplicates are detected, allowing us to use the estimate as pruning method (see Sec. 2.3.4). In practice, variations of the inverse document frequency are used as weight function.

Example 10 *A simpler example for such a weight function is* $w_{rd}(S) = |S|$, *using count as aggregate function. Using this weight function, we obtain* $w_{rd}(N^{\approx}_{rd}(m1,\ m1''))=1$, *and* $w_{rd}(N^{\neq}_{rd}(m1,\ m1''))=4$.

We make analogous definitions to compute the similarity and the difference of ODs, i.e., $N^{\approx}_{od}$, $N^{\neq}_{od}$, and $w_{od}(S)$. Duplicate ODs for $N^{\approx}_{od}$ are detected with a secondary similarity distance, e.g., edit-distance, which does not vary during the iterative phase and which can thus be precomputed.

The final similarity measure template is:

$$sim(c,c') = \frac{\sum_{i\in\{rd,od\}} w_i(N^{\approx}_i(c,c'))}{\sum_{i\in\{rd,od\}} w_i(N^{\neq}_i(c,c')) + w_i(N^{\approx}_i(c,c'))} \tag{2.3}$$

Example 11 *The total similarity of m1 and m1', which we considered in the previous examples as well, equals* $sim(m1,m2) = \frac{1+0}{4+1+0+0} = 0.25$. *Note that the zeros are due to the empty ODs.*

The similarity measure obtains a result between 0 and 1. Given a user defined similarity threshold θ, if $sim(c,c') > \theta$, c and c' are duplicates, otherwise they are non-duplicates.

Note that in defining $N^{\approx}_{rd}$ and $N^{\neq}_{rd}$, we assume that we know that n_1 and n_2 are duplicates or non-duplicates. Clearly, this knowledge is acquired during the iterative phase, and the similarity measure is defined to increase as this knowledge is gained (based on Eq. 1 and Eq. 2). On the other hand, the similarity and the difference of ODs can be precomputed, as they do not vary during the iterative phase.

Proof. We prove that the similarity of two candidates c and c' monotonously increases with increasing duplicates found in their ODs and RDs, assuming that the properties of the weight function defined in Eq. 2.1 and Eq. 2.2 hold, and $WAgg = sum()$. Let $N^{\neq}_{rd,i} = \{(n_1,\bot),(n_2,\bot),(\bot,n'_1)(\bot,n'_2),\ldots)\}$ and $N^{\approx}_{rd,i} = \{(n_3,n'_3),\ldots\}$ at a given processing point i of a comparison algorithm. In the next step, the algorithm determines that n_1 and n'_1 are duplicates, hence $N^{\neq}_{rd,i+1} = \{(n_2,\bot),(\bot,n'_2),\ldots)\}$ and $N^{\approx}_{rd,i+1} = \{(n_3,n'_3),(n_1,n'_1),\ldots\}$. We prove that $sim_i(c,c') \leq sim_{i+1}(c,c')$ by showing that the nominator of sim increases from step i to step $i+1$, whereas the denominator decreases.

It is true that $N^{\approx}_{rd,i} \subset N^{\approx}_{rd,i+1}$, so using Eq. 2.1 and the fact that $sum(S) \geq sum(S')$ for $S' \subset S$, we conclude that

$$sum\left[w_{rd}(N^{\approx}_{rd,i}), w_{od}(N^{\approx}_{od})\right] \leq sum\left[w_{rd}(N^{\approx}_{rd,i+1}), w_{od}(N^{\approx}_{od})\right]$$

This proves that the nominator of $sim_i(c,c)$ is smaller than the nominator of $sim_{i+1}(c,c)$. The denominator of sim decreases from step i to step $i+1$:

$$\begin{aligned}
& sum\left[w_{rd}(N^{\approx}_{rd_i+1}), w_{od}(N^{\approx}_{od}), w_{rd}(N^{\neq}_{rd_i+1}), w_{od}(N^{\neq}_{od})\right] \\
= \ & sum\left[w_{rd}(N^{\approx}_{rd_i} \setminus \{(n_1,\bot),(\bot,n'_1)\}), w_{od}(N^{\approx}_{od}), w_{od}(N^{\neq}_{rd_i} \cup \{n_1,n'_1\}), w_{od}(N^{\neq}_{od})\right] \\
\leq \ & sum\left[w_{rd}(N^{\approx}_{rd_i}), w_{od}(N^{\approx}_{od}), w_{rd}(N^{\neq}_{rd_i}), w_{od}(N^{\neq}_{od})\right]
\end{aligned}$$

because, following Eq. 2.2, $w(\{(n,n')\}) - sum\left[w(\{(n,\bot)\}), w(\{(\bot,n')\})\right] \leq 0$. The nominator of $sim(c,c')$ increases from step i to step $i+1$, and the denominator decreases from step i to step $i+1$, so we conclude that $sim_i(c,c') \leq sim_{i+1}(c,c')$.

2.3.2 SQL-Based Similarity Computation

To compute a similarity conforming to the template on data not residing in main memory, a straightforward method is to use SQL queries to determine the operands of Eq. 2.3. These include computing the similarity and the difference of ODs and of influencing candidate pairs. As the techniques to compute these are similar, we focus the discussion on ODs and make some remarks concerning influencing candidate pairs.

Figure 2.5 outlines the query determining OD weights. For every pair of candidates (c, c') of type t, we first determine the set SIM of similar OD attributes between these pairs, and the associated weight. Because an OD is usually composed of several types of attributes, similar ODs have to be determined for *every* attribute $a_i \in A_t$ and are unified to obtain the complete set $N_{od}^{\approx}(c, c')$ (ln. 5). Next, the set DIFF = $N_{od}^{\neq}(c, c')$ is determined by selecting all OD attributes from $OD(c)$ and $OD(c')$ that are not in SIM, together with associated weights. Again, different types of attributes are treated individually and unified. The weight aggregation function w_{od} is then applied to SIM (ln. 19) and on DIFF (ln. 20) and the two resulting double values, i.e., $w_{od}(N_{od}^{\approx}(c, c'))$ and $w_{od}(N_{od}^{\neq}(c, c'))$, are returned by the query. COALESCE is used to account for the case where SIM or DIFF are empty, in which case the weight is set to 0.

```
WITH SIM AS (
    SELECT A1, A2, W                                         1
    FROM ODSIMW_a1_t S, C_t C1, C_t C2                       2
    WHERE S.A1 = C1.a_1 AND S.A2 = C2.a_1                    3
        AND C1.CID = c AND C2.CID = c'                       4
    UNION ... a_2 ... UNION ... UNION ... a_n ... ),         5
DIFF AS (                                                    6
    SELECT C1.a_1 A A1.W W                                   7
    FROM C_t C1, ODW_a1_t A1                                 8
    WHERE C1.CID = c                                         9
        AND A1.A = C1.a_1                                   10
        AND A1.A NOT IN (SELECT A1 FROM SIM)                11
    UNION                                                   12
    SELECT C2.a_1 A A2.W W                                  13
    FROM C_t C2, ODW_a1_t A2                                14
    WHERE C2.CID = c'                                       15
        AND A2.A = C2.a_1                                   16
        AND A2.A NOT IN (SELECT A2 FROM SIM)                17
    UNION ... a_2 ... UNION ... UNION ... a_n ...),         18
SW AS (SELECT COALESCE(w_od(W),0) ODSIM FROM SIM),          19
DW AS (SELECT COALESCE(w_od(W),0) ODDIFF FROM DIFF)         20
SELECT ODSIM, ODDIFF FROM SW, DW                            21
```

Figure 2.5: Computing OD weights in SQL

Computing the weights of influencing candidate pairs is analogous to computing OD weights. We first determine the set of influencing pairs $I(c, c')$ of candidate pair (c, c') by selecting all tuples from table *DEP* where *S1*= c and *S2*= c'. Due to the fact that we do not have

multiple types of dependencies, no UNION operation is required. The set of duplicate influencing pairs *SIM*= $N^{\approx}_{rd}(c, c')$ is defined as the set of influencing neighbors in *DEP* where *STATUS_T*= 1, i.e., where the target pair (*T1*,*T2*) is a duplicate. Non-duplicates *DIFF*= $N^{\neq}_{rd}(c, c')$ are all influencing candidates of $I(c, c')$ that do not appear in any tuple of *SIM*. The weights of these sets are aggregated as for OD weights (see Fig. 2.5), using the weight function w_{rd}.

Using SQL queries to compute weights of ODs and influencing pairs has the benefit that the DBMS deals with computing these values for large amounts of data, making use of its optimization capabilities. However, the applicability of the query is limited to aggregate functions supported by the DBMS for the weight functions w_{od} and w_{rd}. As a consequence, we define hybrid similarity computation, where SQL queries are used to gather data, and aggregation is performed in an external program.

2.3.3 Hybrid Similarity Computation

In the hybrid version of OD weight computation, we use two queries Q_1 and Q_2. Q_1 determines the set of similar OD attribute pairs with their weight, corresponding to the subquery SIM in Fig. 2.5. Q_2 determines the set of *all* OD attributes defined as $OD(c) \cup OD(c')$, so it is essentially the subquery DIFF in Fig. 2.5 without line 11 and line 17, which check whether an attribute value is in the set of similar attribute values. In hybrid similarity computation, this check is performed outside the database in an external program, as well as weight aggregation. The external program consists of the following steps, where the sets D (duplicates) and N (non-duplicates) are initially empty.

1. For every tuple, $< v_1, v_2, w >$ returned by Q_1, let $D := D \cup \{((v_1, v_2), w)\}$.
2. For every tuple $< v, w >$ returned by Q_2, check if $\{(v_{1,i}, v_{2,i}) | (v_{1,i}, v_{2,i}) \in D \wedge (v_{1,i} = v \vee v_{2,i} = v)\} = \emptyset$. If it is empty, let $N := N \cup \{((v, \perp), w)\}$.
3. Once all pairs have been processed, compute the aggregate weights $w_{od}(D)$ and $w_{od}(N)$.

Example 12 *When computing OD similarity and OD difference of (*a1, a1'*), Q_1 returns similar values pairs, which are added to D in Step 1. Hence, $D = \{(($*Brad Pitt, Brad Pit*$), 1)\}$. Q_2 returns $OD($*a1*$) \cup OD($*a1'*$) = \{($*Brad Pitt*$, 1), ($*Brad Pit*$, 1)\}$. Because both OD values are part of a similar pair in D, $N = \emptyset$ in Step 2. Step 3 computes $w_{od}(D) = 1$ and $w_{od}(N) = 0$, assuming w_{od} computes the sum of weights.*

For influencing pairs, the hybrid strategy is slightly different. Indeed, for ODs we have only a precomputed table for similar OD attribute values, so Q_1 and Q_2 have different input tables. For influencing pairs, table DEP stores both duplicate influencing pairs and non-duplicate influencing pairs together with their status. Consequently, we use a query Q_3 to determine $I(c, c')$, which sorts influencing pairs in the ascending order of their status, so that duplicate pairs are the first pairs in the result. The external program then takes care of splitting up duplicates and non-duplicates. The order chosen guarantees that all duplicates are added to D before the first non-duplicate appears, so that we can apply Step 2 as for OD weight computation.

Example 13 *When comparing (m1, m1'), Q_3 returns tuples {(a1,a1',1,1), (a2,a2',1,1), (a1,a3,1,0), (a2,a1',1,0), (a2,a3,1,0), (t1,t2,1,0)} in that order, where the tuple schema is <cand1, cand2, weight, status>. The external program iterates through these tuples and adds them to D as long as status equals 0. This results in D = {((a1,a1'),1), ((a2,a2'),1) }. The remaining tuples are split up in two candidates and we apply Step 2 and obtain N = {(((a3),⊥),1), (((t1),⊥),1), (((t2),⊥),1)}. Using weight sum as w_{rd}, we obtain $w_{rd}(D) = 2$ and $w_{rd}(N) = 3$.*

Compared to the SQL based similarity computation, we have to send three queries to the database instead of one query, which can be a drawback due to communication overhead. Furthermore, the results returned by Q_1, Q_2, and Q_3 are larger than the result of the query used in SQL based similarity computation, and these results need to be processed in main memory, i.e., entries are added to D and N and are aggregated. But compared to the in-memory buffer, we consider this main-memory consumption as negligible, because in the worst case $|D|+|N| = |I(c)|+|I(c')|$, which can be taken into account when setting the buffer size. In practice, these sets are small (14 candidates being the maximum observed in experiments reported in DDG algorithms summarized in Sec. 1.2). The main advantage of hybrid similarity computation is that it overcomes database system limitations, e.g., when weight aggregation functions are not supported. We expect that both methods have comparable runtime, because they both compute the same, only that processing is split between the database and an external program for hybrid similarity computation. This is confirmed by our experiments. We now present a technique that improves classification runtime when using hybrid similarity computation.

2.3.4 Early Classification

Essentially, early classification interrupts similarity computation as soon as we know if the outcome results in a duplicate or non-duplicate classification. Early classification distinguishes itself from existing filters defined as upper bounds to the similarity measure [1, 23] in that no extra filter function is defined to prune non-duplicates prior to similarity computation. Instead, the similarity function is computed incrementally and intermediate results are used to classify non-duplicates or duplicates prior to termination. As real-world data usually contains only a small percentage of duplicates, we decide to determine non-duplicates more efficiently using early classification, although early classification can also be used to classify pairs of candidates as duplicates before similarity has been completely calculated.

If a candidate pair similarity $sim(c,c') > \theta$, candidates c and c' are duplicates, and non-duplicate otherwise. Hence, the following inequations, which we use in our implementation, correctly classify non-duplicates, although sim (Eq. 2.3) is not calculated completely. As a reminder, sim is defined as

$$sim(c,c') = \frac{\sum_{i\in\{rd,od\}} w_i(N_i^{\approx})}{\sum_{i\in\{rd,od\}} w_i(N_i^{\neq}) + w_i(N_i^{\approx})} \tag{2.4}$$

Parameters c and c' of sim, $N_{od}^{\approx}$, $N_{rd}^{\approx}$, and $N_{rd}^{\neq}$ have been omitted for brevity. It is easy to verify that the following holds:

$$w_{od}(N_{od}^{\approx}) + w_{rd}(N_{rd}^{\approx}) \leq \theta \rightarrow sim \leq \theta \tag{2.5}$$

$$\frac{w_{od}(N_{od}^{\approx}) + w_{rd}(N_{rd}^{\approx})}{w_{rd}(N_{rd}^{\neq}) + w_{od}(N_{od}^{\approx}) + w_{rd}(N_{rd}^{\approx})} \leq \theta \rightarrow sim \leq \theta \quad (2.6)$$

To include early classification, we modify hybrid similarity computation. First, we do not treat ODs and influencing pairs sequentially by first computing the similarity and the difference of ODs, and then computing them for influencing pairs. Instead, we compute their similarity to obtain $N_{od}^{\approx} + N_{rd}^{\approx}$, and then incrementally compute the difference of ODs and influencing pairs. That is, we execute Q_1 and Q_3 and perform Step 1 for ODs and iterate through the result of Q_3 until the duplicate status stored in the attribute *STATUS* of a tuple switches to 0. Before we continue, we apply Eq. 2.5. If it classifies the pair as a non-duplicate, we avoid iterating over non-duplicates in Q_3 and executing Q_2. Otherwise, we start computing the difference of influencing pairs by iterating through the remaining tuples returned by Q_3. At every iteration, we check if Eq. 2.6 can classify the pair as non-duplicate to potentially save further iteration as well as query Q_2. If the pair is not classified as a non-duplicate after iterating through Q_3's result, we execute Q_2 and iterate over its result, again checking at every iteration if Eq. 2.6 classifies a non-duplicate. When reaching the final iteration, we have finally computed $sim(c, c')$, and return the corresponding classification result. This implementation of early classification guarantees that similarity computation stops as soon as one of above rules classifies a pair as non-duplicate.

Using early classification helps to detect non-duplicates without computing the exact similarity. So it helps to increase classification efficiency when a significant portion of pairs are non-duplicates. The larger the number of influencing pairs or OD attribute values of candidates are, the more processing is potentially saved using early classification, because the number of iterations through non-duplicates among influencing pairs and ODs (i.e., in the results of Q_3 and Q_2, when it was executed) that are potentially saved increases the larger the set of influencing pairs and ODs get. This explains why in our experiments, which are discussed next, classification time is not affected by increasing the number of influencing neighbors (and hence pairs) when using early classification.

```
/* Retrieval                                                        */
while executeSQL(
   SELECT COUNT(*) C FROM PQT WHERE STATUS = 0
) returns C > 0 do
    boolean isDup ← F ;
    ResultSet r ← executeSQL(
    SELECT * FROM PQT WHERE STATUS = 0 ORDER BY RANK  );
    while r has more tuples and isDup = F do
        Tuple t ← r.getNextTuple();
        /* Classification                                           */
        isDup ←
           classify(t.getValue(C1), t.getValue(C2));
        /* Update                                                   */
        if isDup = true then
            t.updateValue(STATUS, 1);
            cluster(t.getValue(C1), t.getValue(C2));
            executeSQL(
                UPDATE REL SET STATUS2 = 1
                WHERE T1 = t.getValue(C1)
             AND T2 = t.getValue(C2) ) ;
                ResultSet d ← executeSQL(
                    SELECT S1, S2, RANK(S1,S2) R FROM DEP
                    WHERE T1 = t.getValue(C1)
                    AND T2 = t.getValue(C2)
                 AND STATUS1 ≠ 0 OR 1 ) ;
                    while d has more tuples do
                        executeSQL(
                            UPDATE PQT
                            SET STATUS = 0, RANK = d.getValue(R)
                            WHERE C1 = d.getValue(T1)
                         AND C2 = d.getValue(T2) ) ;
        else
            t.updateValue(STATUS,-1)
```

Algorithmus 1 : **RECUS/DUP Algorithm**

```
while PQT or B_s has unclassified pairs do
    boolean overflow ← false;
    boolean lastFromPQ ← true;
    boolean updateOnly ← false;
    ResultSet r ← executeSQL( same query as in Alg. 1 );
    while r has more tuples and overflow = false do
        Tuple t;
        if lastFromPQ = true then
            t ← r.getNextTuple();
            if t ∈ B_s then
                t.updateValue(STATUS,status in B_s);
                updateOnly ← true;
            else
                updateOnly ← false;
        else
            t.getCurrentTuple();
            lastFromPQ ← true;
        Pair tb := b.getFirstEntry();
        if updateOnly = false and pb.getRank() ≤ p.getRank() then
            p ← pb;
            lastFromPQ ← false;
        else
            lastFromPQ ← true;
        ... classification & update ...
```

Algorithmus 2 : **RECUS/BUFF Retrieval Phase**

Chapter 3

Evaluation

We evaluate our proposed methods based on experiments on artificial data, experiments on real-world data, and a comparative study to related work.

3.1 Data Sets

RealCD (Real-world CD Data): The real-world data set we use comes from the CD domain [1]. We consider CDs, artists, and track titles as candidates. CD ODs consist of title, year, genre, and category attributes, and they are related to artist and track candidates. Artist and track candidates respectively have the artist's name and the track title as OD. Track candidates depend on artist candidates. Using RealCD, we have little influence on data characteristics. To modify parameters and study their effect, we also use artificial data.
ArtMov (Artificial Movie Data): From a list of 35,000 movie names and 800,000 actors from IMDB[2], we generate data sets for which we control (i) the number of candidate movies M and the number of candidate actors A to be generated, (ii) the connectivity c, i.e., the average number of actors that influence a movie and vice versa, (iii) the duplicate ratio dr, defined as the percentage of duplicate pairs that enter the priority queue, (iv) the probability of errors in ODs consisting of a movie name and an actor name for the respective candidate types, and (v) the probability of errors in influencing candidates. Further details are available in [31].

In our experiments, $A = M = \frac{k}{2}$. The number of duplicate pairs equals $k * dr$, which result from duplicating $dr\frac{k}{2}$ original movie candidates as well as $dr\frac{k}{2}$ actor candidates. The remaining $k - k * dr$ non-duplicate pairs are equally distributed between movie pairs and actor pairs, each resulting from x non duplicates such that $k - dr * k = x * (x - 1)$. Hence, the relationship between number of candidates and priority queue size is given by

$$A = M = max\left(\frac{pq * dr}{2}, roundUp\left(x\right)\right) + \frac{pq * dr}{2} \tag{3.1}$$

In all our experiments error probabilities are set to 20%. When not mentioned otherwise, connectivity $c = 5$ and the buffer's size is set to 1,000. We repeated experiments five times to obtain

[1] http://www.freedb.org
[2] http://www.imdb.com

average execution times, which, in addition to the large number of experiments explains the moderate number of candidates for ArtMov data compared to the real-world data set.

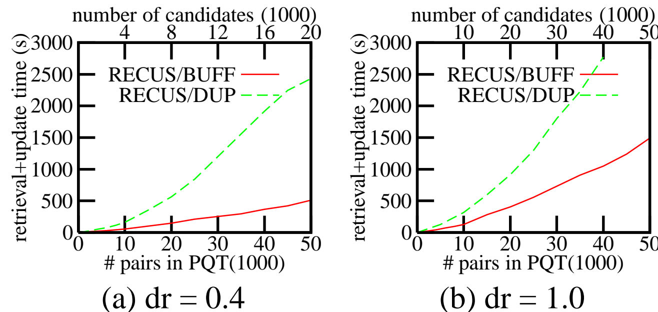
3.2 Experimental Evaluation

All presented experiments used DB2 V8.2 as DBMS, running on a Linux server and remotely accessed by a Java program running on a Pentium 4 PC (3.2GHz) with 2GB of RAM. That is, all runtimes reported also include network latency.

3.2.1 Retrieval and Update Scalability

We first compare the scalability of RECUS/BUFF to the scalability of the baseline algorithm RECUS/DUP.

Experiment 1. We start with an evaluation of how both algorithms behave with varying dr on various data set sizes.
Methodology. We generate ArtMov data according to Eq. 3.1 with pq ranging from 10,000 to 50,000 candidate pairs in increments of 5,000, and vary the duplicate ratio dr between $dr = 0.2$ and $dr = 1.0$ in increments of 0.2. As representative results, we show runtimes of RECUS/DUP and RECUS/BUFF (in seconds) for $dr = 0.4$, and $dr = 1.0$ in Fig. 3.1. The number of candidates ($|A| + |M|$) is shown at the top x-axis, whereas the bottom x-axis shows the size of *PQT*(in thousands). Both axes are correlated through Eq. 3.1.



(a) dr = 0.4 (b) dr = 1.0

Figure 3.1: Retrieval & update time varying data set size & dr.

Discussion. Fig. 3.1 clearly shows that RECUS/BUFF outperforms RECUS/DUP regardless of dr. Obviously, sorting *PQT* is more time consuming than maintaining the order of the smaller in-memory buffer. We further observe that the higher dr, the more time retrieval and update need for both algorithms, because in both algorithms sorting *PQT* occurs more frequently: RECUS/DUP sorts *PQT* every time a duplicate has been found, and RECUS/BUFF sorts the *PQT* every time the buffer overflows, happening more frequently, because influencing neighbor pairs enter B_s more frequently. The final observation is that with increasing priority queue size / number of candidates, RECUS/BUFF scales almost linearly for practical duplicate ratios (below 0.8). Therefore, we expect RECUS/BUFF to be efficient even on very large data sets such as RealCD, as Exp. 6 confirms.

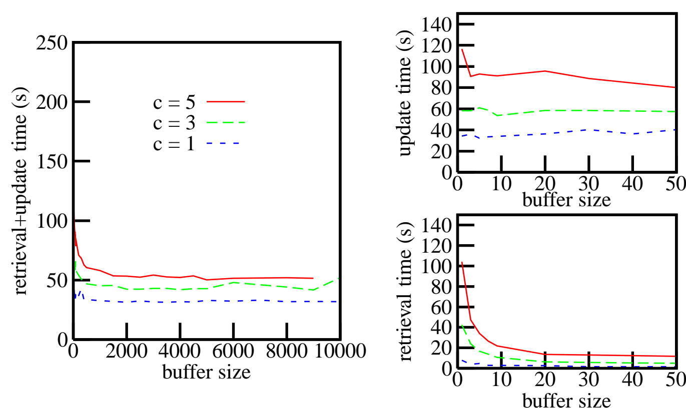


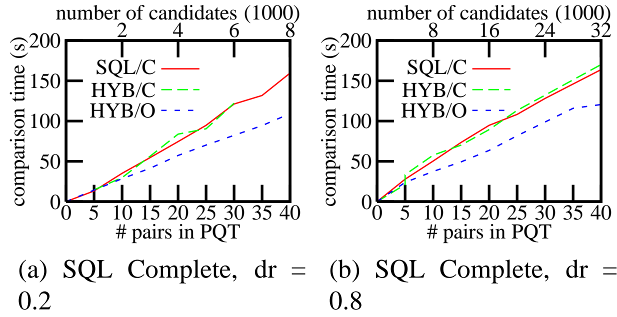
Figure 3.2: Retrieval & update time for varying s and c.

Experiment 2. From Exp. 1, we conclude that RECUS/BUFF performs better than RECUS/DUP, because it sorts *PQT* less frequently, and instead maintains a main-memory buffer of fixed size s. Clearly, s plays a central role in the efficiency gain. Another factor that affects the filling of B_s is the connectivity c. The higher it is, the more neighbors enter the buffer when a duplicate is found, and an overflow occurs more frequently. So, we expect that RECUS/BUFF gets slower with increasing c and smaller s.
Methodology. We study how a changing buffer size affects ArtMov data with 10,000 pairs in *PQT* and a $dr = 0.4$. We vary s from 1 to 10,000. We further vary connectivity c from 1 to 5 for all considered buffer sizes. Results are shown in Fig. 3.2, which depicts the sum of retrieval and update time for all buffer sizes (left), update time for small buffer sizes (top), and retrieval time for small buffer sizes (bottom).
Discussion. We observe that for all but small buffer sizes, both retrieval time and update time stabilize, and hence the sum stabilizes. For very small buffer sizes, the smaller the buffer, the longer retrieval and update take. Furthermore, the larger the connectivity c, the more time is needed for retrieval and update, mainly resulting from the increased update complexity. For larger c, when a duplicate is found, a larger number of dependent pairs needs to be determined and added to the buffer. As a general rule of thumb, a buffer size of 1000 suffices to significantly improve efficiency.

3.2.2 Classification Scalability

We evaluate the scalability of classification, considering the baseline SQL based approach, we call SQL/Complete (SQL/C for short) and hybrid similarity computation with and without early classification, called HYB/Complete (HYB/C) and HYB/Optimized (HYB/O), respectively.
Experiment 3. We compare runtimes of SQL/C, HYB/C, and HYB/O on ArtMov data of different sizes, using varying duplicate ratios. We expect SQL/C to be slower than HYB/O, whereas SQL/C should be comparable to HYB/C.
Methodology. We generate ArtMov data with *PQT* sizes ranging from 5,000 to 40,000 in increments of 5,000, and for each size, we generate data with dr varying from 0.2 to 1.0 in increments of 0.2. We run each classification method on each data set, and measure its runtime

in seconds. Results are shown in Fig. 3.3 for selected duplicate ratios dr = 0.2, and dr = 0.8.



(a) SQL Complete, dr = 0.2

(b) SQL Complete, dr = 0.8

Figure 3.3: Classification time comparison

Discussion. As expected, SQL/C and HYB/C have comparable execution times, because they both have to compute the same result. All classification methods scale linearly on the range of considered priority queue sizes, and hence with the number of candidates when no blocking technique is additionally used. Early classification allows to save classification time: at dr = 0.2, 32 % of classification time (compared to HYB/C) is saved, which gracefully degrades to 26% at dr = 0.8, when 40,000 pairs are compared. For dr = 1, we still observe 5% savings, which are due to pairs that are classified as non-duplicates, although they are (false negatives). The reduction in the benefit of early classification with increasing dr is due to the fact that the more duplicates are in *PQT*, the less similarity computations may be aborted for non-duplicates.

Experiment 4. Our next experiment shows how the connectivity c affects classification efficiency. Because c defines how many influencing neighbors a candidate has, which have to be determined using join operations and processed, we expect similarity computation to be slower for larger c when using SQL/C or HYB/C. On the other hand, HYB/O potentially saves more processing the larger c.

Methodology. We vary c, i.e., the average number of influencing candidates, from c = 10 to c = 50 in increments of 10 for an ArtMov dataset of size 10,000 and duplicate ratio 0.4. Figure 3.4 reports comparison time for SQL/C and HYB/O, SQL/C and HYB/C being comparable.

Discussion. When using SQL/C, comparison time increases with increasing c, an effect also observed by other DDG algorithms. On the other hand, runtime is around 7 ms for all c when using HYB/O. This experiment shows that HYB/O counters the negative effect of increasing c on efficiency. We currently do not have an explanation for the shape of the SQL/C curve, where comparison time is roughly constant between c = 20 and c = 40. We suspect that "intriguing behavior of modern [query] optimizers" [26] is partly responsible for that behavior, but this needs further investigation. Nevertheless, the general trend is clear.

The analysis of the techniques proposed in this paper using artificial data of moderate size leads to the conclusion that RECUS/BUFF and HYB/O scale up best. We now put them to the test by applying them to large, real-world data.

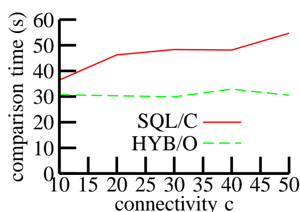


Figure 3.4: Classification time & connectivity

3.2.3 Real-World Behavior

Using RealCD, we show real-world scalability of our approach over a large data set. Results on effectiveness are omitted due to the lack of space. The benefit of DDG on effectiveness has already been studied extensively (Sec. 1.2).

Experiment 5. We show that RECUS/BUFF in combination with HYB/O scales linearly with the number of candidate pairs in *PQT* on a large, real-world data set.

Methodology. Using RealCD data with 1,000,000 candidates, we apply a blocking technique to reduce the number of candidate pairs entering *PQT*, a common technique also used by [6, 19]. Note that we evaluate the behavior of our approaches according to the number of candidate pairs that are in *PQT*, so this does not affect our conclusions. After blocking, 2,000,000 candidate pairs enter *PQT*. Buffer size is 1,000, so that we can compare runtimes with those obtained on artificial data. We obtain the following results: retrieval takes 1,379 s, classification takes 5,482 s, and update takes 17,572 s.

Discussion. Among the two million candidate pairs in *PQT*, we found 600,000 duplicates, so the observed duplicate ratio is 0.3. If we extrapolate retrieval and update time obtained on ArtMov data of size 50,000 with $dr = 0.3$, for which we obtained a retrieval time of 36 seconds and 432 seconds for update using a buffer of 1,000, we see that the results obtained on a million candidates with similar parameters are in accord with the linear behavior of retrieval and update observed in Exp. 1. Indeed, the expected retrieval time is 1,440 seconds, so the observed 1,379 seconds are 4% from the expected value. Similarly, the expected 17,280 seconds for update and the 17,572 seconds measured are only 2% apart. Classification time is also in accord with the linear behavior of HYB/O observed in Exp. 3.

3.3 Comparative Evaluation

Table 3.1(a) summarizes how the different phases of DDG scale in time depending on the size of the data s, the duplicate ratio dr, and the connectivity c, which in total amounts to a linear behavior. Table 3.1(b) summarizes results *reported* for other DDG algorithms, omitting those that do not report any runtime, which altogether use smaller data sets as those reported here. Tab. 3.1(b) reports on the data set size, runtime (without initialization time) and the parameters for which the algorithms do *not* scale linearly (s, dr, and c are considered). We observe that RECUS/BUFF takes comparably long, but this comes as no surprise as DB communication overhead and network latency add to the runtime. More interestingly, none of the DDG algorithms except RECUS/BUFF scales linearly in time with all three parameters s, dr, and c.

Indeed, all algorithms but RECUS/BUFF do not scale linearly in time with the data set size s, which compromises scaling up DDG to large amounts of data.

Parameter	Retrieval & update	Classification
PQT size s	linear (Exp. 1,5)	linear (Exp. 3,5)
duplicate ratio $dr < 0.8$	linear (Exp. 1)	constant (Exp. 3)
connectivity c	linear (Exp. 2)	constant (Exp. 4)
Overall	**linear**	**linear**

(a) DDG scalability using RECUS/BUFF and HYB/O

Approach	# candidates	Runtime (s)	Not linear in
RC-ER [6]	68,000	890	s, c
RelDC [19]	75,000	180 - 13,000 [a]	s, c
LinkClus [33]	100,000	900	s
RECUS/BUFF	**1,000,000**	**24,433**	-

[a] depending on connectivity

(b) Comparison time for different approaches

Table 3.1: Comparative evaluation

Chapter 4

Conclusion and Outlook

This paper is the first to consider scalability of duplicate detection in graphs (DDG). We presented a generalization of existing iterative DDG algorithms consisting of an initialization phase and an iterative phase. The latter in turn consists of retrieval, classification, and update steps. We then presented how to scale up these phases to large amounts of data, with the help of an RDBMS.

For iterative retrieval and update, we proposed RECUS/BUFF to scale in space and in time. It uses an internal buffer to avoid expensive sorting, which is performed by the straightforward baseline algorithm RECUS/DUP.

To scale up classification of candidates, we proposed hybrid similarity computation to scale in space and to overcome the limitations of a pure SQL variant. To scale up classification in time, we presented the early classification technique, which interrupts similarity computation when it is sure that a pair is not a duplicate.

Experiments on large amounts of data, such as one million candidates (at least an order of magnitude larger than previously considered data sets) validate our approaches and show that the methods we propose significantly outperform a straightforward mapping of DDG from main-memory to a database. Part of the research presented here was successfully applied to an industry project [32]. More specifically, we used the similarity measure template to systematically vary different similarity-based classifiers. Finally, our research on scaling classification yielded to the design decision of using hybrid similarity measurement with early classification.

In the future, we plan to detect a candidate's OD attributes and influencing candidate types automatically, because this is the task requiring the most user interaction. Instead of determining duplicates in a data source, we also plan to investigate how we can avoid duplicates to enter the database.

Bibliography

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB Conference*, Hong Kong, China, 2002.

[2] I. Bhattachary, L. Getoor, and L. Licamele. Query-time entity resolution. In *KDD Conference*, Philadelphia, PA, 2006. Poster.

[3] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. *SIGMOD DMKD Workshop*, 2004.

[4] I. Bhattacharya and L. Getoor. Entity resolution in graph data. Technical Report CS-TR-4758, University of Maryland, 2006.

[5] I. Bhattacharya and L. Getoor. A latent Dirichlet model for unsupervised entity resolution. In *SDM Conference*, Bethesda, MD, 2006.

[6] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data*, 1(1), March 2007.

[7] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD Conference*, Washington, DC, 2003.

[8] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE Conference*, 2006.

[9] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE Conference*, Tokyo, Japan, 2005.

[10] Z. Chen, D. V. Kalashnikov, and S. Mehrotra. Exploiting relationships for object consolidation. In *SIGMOD IQIS Workshop*, Baltimore, MD, 2005.

[11] A. Doan, Y. Lu, Y. Lee, and J. Han. Object matching for information integration: A profiler-based approach. *IEEE Intelligent Systems*, 2003.

[12] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, Baltimore, MD, 2005.

[13] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1), 2007.

[14] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 1969.

[15] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB Conference*, Rome, Italy, 2001.

[16] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB Conference*, Roma, Italy, 2001.

[17] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD Conference*, San Jose, CA, 1995.

[18] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *DASFAA Conference*, Kyoto, Japan, 2003.

[19] D. V. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Trans. Database Syst.*, 31(2), 2006.

[20] D. Milano, M. Scannapieco, and T. Catarci. Structure aware xml object identification. In *VLDB CleanDB Workshop*, 2006.

[21] A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *SIGMOD DMKD Workshop*, Tuscon, AZ, 1997.

[22] H. Newcombe, J. Kennedy, S. Axford, and A. James. Automatic linkage of vital records. *Science 130 (1959) no. 3381*, 1959.

[23] B.-W. On, N. Koudas, D. Lee, and D. Srivastava. Group linkage. In *ICDE Conference*, 2007.

[24] S. Puhlmann, M. Weis, and F. Naumann. XML duplicate detection using sorted neigborhoods. *EDBT Conference*, 2006.

[25] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin, Volume 23*, 2000.

[26] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB Conference*, Trondheim, Norway, 2005.

[27] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD Conference*, Edmonton, Alberta, 2002.

[28] P. Singla and P. Domingos. Object identification with attribute-mediated dependences. In *Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, Porto, Portugal, 2005.

[29] M. Weis and F. Naumann. Dogmatix tracks down duplicates in XML. In *SIGMOD Conference*, Baltimore, MD, 2005.

[30] M. Weis and F. Naumann. Detecting duplicates in complex XML data. In *ICDE Conference*, Atlanta, Georgia, 2006. Poster.

[31] M. Weis and F. Naumann. Relationship-based duplicate detection. Technical Report HU-IB-206, Humboldt University Berlin, 2006.

[32] M. Weis and F. Naumann. Industry-scale duplicate detection (to appear). In *VLDB Conference*, 2008.

[33] X. Yin, J. Han, and P. S. Yu. LinkClus: Efficient clustering via heterogeneous semantic links. In *VLDB Conference*, 2006.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
24	978-3-940793-45-4	**Erster Deutscher IPv6 Gipfel**	Christoph Meinel, Harald Sack, Justus Bross
23	978-3-940793-42-3	**Proceedings of the 2nd. Ph.D. retreat of the HPI Research School on Service-oriented Systems Engineering**	Alle Professoren des HPI
22	978-3-940793-29-4	**Reducing the Complexity of Large EPCs**	Artem Polyvyanyy, Sergy Smirnov, Mathias Weske
21	978-3-940793-17-1	**"Proceedings of the 2nd International Workshop on e-learning and Virtual and Remote Laboratories"**	Bernhard Rabe, Andreas Rasche
20	978-3-940793-02-7	**STG Decomposition: Avoiding Irreducible CSC Conflicts by Internal Communication**	Dominic Wist, Ralf Wollowski
19	978-3-939469-95-7	**A quantitative evaluation of the enhanced Topic-based Vector Space Model**	Artem Polyvyanyy, Dominik Kuropka
18	978-939-469-58-2	**Proceedings of the Fall 2006 Workshop of the HPI Research School on Service-Oriented Systems Engineering**	Benjamin Hagedorn, Michael Schöbel, Matthias Uflacker, Flavius Copaciu, Nikola Milanovic
17	3-939469-52-1 / 978-939469-52-0	**Visualizing Movement Dynamics in Virtual Urban Environments**	Marc Nienhaus, Bruce Gooch, Jürgen Döllner
16	3-939469-35-1 / 978-3-939469-35-3	**Fundamentals of Service-Oriented Engineering**	Andreas Polze, Stefan Hüttenrauch, Uwe Kylau, Martin Grund, Tobias Queck, Anna Ploskonos, Torben Schreiter, Martin Breest, Sören Haubrock, Paul Bouché
15	3-939469-34-3 / 978-3-939469-34-6	**Concepts and Technology of SAP Web Application Server and Service Oriented Architecture Products**	Bernhard Gröne, Peter Tabeling, Konrad Hübner
14	3-939469-23-8 / 978-3-939469-23-0	**Aspektorientierte Programmierung – Überblick über Techniken und Werkzeuge**	Janin Jeske, Bastian Brehmer, Falko Menge, Stefan Hüttenrauch, Christian Adam, Benjamin Schüler, Wolfgang Schult, Andreas Rasche, Andreas Polze
13	3-939469-13-0 / 978-3-939469-13-1	**A Virtual Machine Architecture for Creating IT-Security Labs**	Ji Hu, Dirk Cordel, Christoph Meinel